Distributed and Parallel Databases 5, 233–269 (1997)

Adaptive Algorithms for Join Processing in Distributed Database Systems

PETER SCHEUERMANN* peters@ece.nwu.edu
Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208

EUGENE INSEOK CHONG† echong@us.oracle.com
New England R&D Center, Oracle Corporation, 1 Oracle Drive, Nashua, NH 03062

Received May 10, 1996; Accepted February 19, 1997

Recommended by: Clement Yu

Abstract. Distributed query processing algorithms usually perform data reduction by using a semijoin program, but the problem with these approaches is that they still require an explicit join of the reduced relations in the final phase. We introduce an efficient algorithm for join processing in distributed database systems that makes use of bipartite graphs in order to reduce data communication costs and local processing costs. The bipartite graphs represent the tuples that can be joined in two relations taking also into account the reduction state of the relations. This algorithm fully reduces the relations at each site. We then present an adaptive algorithm for response time optimization that takes into account the system configuration, i.e., the additional resources available and the data characteristics, in order to select the best strategy for response time minimization. We also report on the results of a set of experiments which show that our algorithms outperform a number of the recently proposed methods for total processing time and response time minimization.

Keywords: distributed query processing, join algorithms, adaptive algorithms, bipartite graphs

1. Introduction

Query processing in distributed database systems often requires the transmission of relations and/or temporary results among different sites via a computer network. Until recently it has been assumed that communication costs were the predominant costs and hence much of the research on query optimization in distributed database systems has concentrated on producing optimal or near-optimal strategies with regard to data transmission costs [1, 3, 5, 6, 8, 12–14].

In order to reduce the communication time, most algorithms for distributed query processing involve three phases [20, 30]: 1) a local processing phase which includes all local operations such as selections and projections, 2) a reduction phase in which an effective sequence of semi-join (and join) operations is further used to reduce the sizes of the relations,

*The work of this author was partially supported by NSF grant IRI-9303583 and NASA-Ames grant NAG2-846.
†The work of this author was performed while he was with Northwestern University.

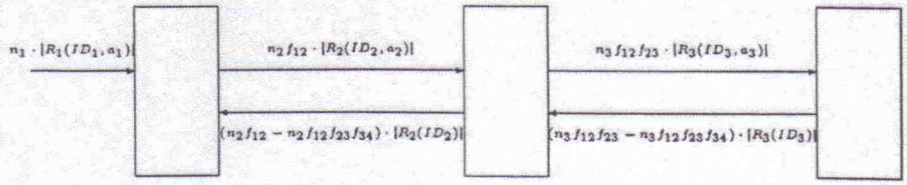

Figure 2. Sizes of transmitted data (PIPE_CHQ).

components. The dominant components of the processing time are the sum $S_{i,f}$, the time to construct the bipartite graphs in the forward reduction phase, and ***GT***, the graph traversal time. Inside the component $S_{i,f}$, the major contribution is the I/O cost for the merge-sort step, while inside ***GT***, the major cost is the I/O time involved in retrieving the target attributes for the tuples in the implicit join. If we denote by $\bar{P}_i$ the average relation size in pages and by K the cardinality of the implicit join, then we obtain that the time complexity of the processing time is bounded by $O(n\bar{P}_i \log_2 \bar{P}_i + nK)$, with n being the number of sites.

2.2. *Disk-based systems*

The previous algorithm and cost model were based on the assumption that the bipartite graphs fit in main memory. We shall show now that our algorithm can be easily extended to disk-based systems where only a portion of each graph is memory resident. As each bipartite graph is constructed it is dynamically partitioned into subgraphs that are stored on secondary storage, i.e., whenever the buffer allocated to the construction of the graph $BG_{R_i,R_{i+1}}$ becomes full its page(s) is(are) flushed to secondary storage. Thus, each page on secondary storage consists of a subgraph. We observe that these subgraphs are not necessarily disjoint, i.e., they may have crossing edges.

In order to minimize the number of I/O operations that are performed on the bipartite graphs during the backward reduction phase, each graph stored at site S_i is augmented so as to include for every edge (id_{i-1}, id_i) also the page number in secondary storage associated with the tuple containing id_{i-1} at S_{i-1}, to be denoted by $page(id_{i-1})$. Thus, we can view now a bipartite graph BG_{R_{i-1},R_i} as corresponding to a normalized triary relation $BG_{R_{i-1},R_i}(ID_{i-1}, ID_i, PAGE(ID_{i-1}))$. Similarly, the messages sent in the forward and backward phases are augmented correspondingly to include page numbers. Hence, forward messages sent from S_i, the site of BG_{R_{i-1},R_i}, are now of the form $(id_i, a_i, \{page(id_i)\})$, with a_i being the value of the join attribute A_i and $\{page(id_i)\}$ standing for the page number(s) in secondary storage associated with the tuples containing id_i at S_i. Backward messages sent from a site S_i have the format $(id_{i-1}, page(id_{i-1}))$. As we shall see below, using these page numbers we can guarantee that during backward reduction each page of a bipartite graph is read and written back to storage only once.

Since the construction of the bipartite graphs during the forward phase does not guarantee that the subgraphs stored on secondary storage are disjoint, at S_i a particular tuple identifier id_i may appear in a number of distinct pages, connected in each to disjoint (sets of) tuple

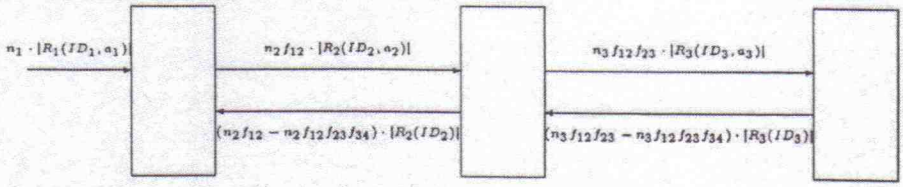

Figure 2. Sizes of transmitted data (PIPE_CHQ).

components. The dominant components of the processing time are the sum $S_{i,f}$, the time to construct the bipartite graphs in the forward reduction phase, and ***GT***, the graph traversal time. Inside the component $S_{i,f}$, the major contribution is the I/O cost for the merge-sort step, while inside ***GT***, the major cost is the I/O time involved in retrieving the target attributes for the tuples in the implicit join. If we denote by $\bar{P}_i$ the average relation size in pages and by K the cardinality of the implicit join, then we obtain that the time complexity of the processing time is bounded by $O(n\bar{P}_i \log_2 \bar{P}_i + nK)$, with n being the number of sites.

2.2. *Disk-based systems*

The previous algorithm and cost model were based on the assumption that the bipartite graphs fit in main memory. We shall show now that our algorithm can be easily extended to disk-based systems where only a portion of each graph is memory resident. As each bipartite graph is constructed it is dynamically partitioned into subgraphs that are stored on secondary storage, i.e., whenever the buffer allocated to the construction of the graph $BG_{R_i,R_{i+1}}$ becomes full its page(s) is(are) flushed to secondary storage. Thus, each page on secondary storage consists of a subgraph. We observe that these subgraphs are not necessarily disjoint, i.e., they may have crossing edges.

In order to minimize the number of I/O operations that are performed on the bipartite graphs during the backward reduction phase, each graph stored at site S_i is augmented so as to include for every edge (id_{i-1}, id_i) also the page number in secondary storage associated with the tuple containing id_{i-1} at S_{i-1}, to be denoted by $page(id_{i-1})$. Thus, we can view now a bipartite graph BG_{R_{i-1},R_i} as corresponding to a normalized triary relation $BG_{R_{i-1},R_i}(ID_{i-1}, ID_i, PAGE(ID_{i-1}))$. Similarly, the messages sent in the forward and backward phases are augmented correspondingly to include page numbers. Hence, forward messages sent from S_i, the site of BG_{R_{i-1},R_i}, are now of the form $(id_i, a_i, \{page(id_i)\})$, with a_i being the value of the join attribute A_i and $\{page(id_i)\}$ standing for the page number(s) in secondary storage associated with the tuples containing id_i at S_i. Backward messages sent from a site S_i have the format $(id_{i-1}, page(id_{i-1}))$. As we shall see below, using these page numbers we can guarantee that during backward reduction each page of a bipartite graph is read and written back to storage only once.

Since the construction of the bipartite graphs during the forward phase does not guarantee that the subgraphs stored on secondary storage are disjoint, at S_i a particular tuple identifier id_i may appear in a number of distinct pages, connected in each to disjoint (sets of) tuple

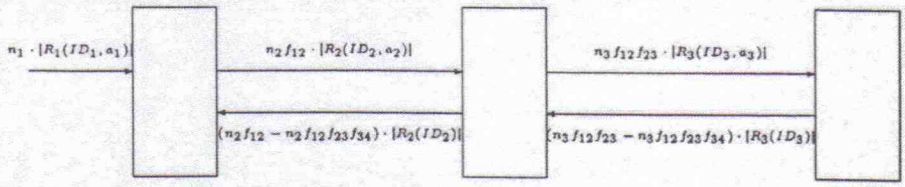

Figure 2. Sizes of transmitted data (PIPE_CHQ).

components. The dominant components of the processing time are the sum $S_{i,f}$, the time to construct the bipartite graphs in the forward reduction phase, and ***GT***, the graph traversal time. Inside the component $S_{i,f}$, the major contribution is the I/O cost for the merge-sort step, while inside ***GT***, the major cost is the I/O time involved in retrieving the target attributes for the tuples in the implicit join. If we denote by $\bar{P}_i$ the average relation size in pages and by K the cardinality of the implicit join, then we obtain that the time complexity of the processing time is bounded by $O(n\bar{P}_i \log_2 \bar{P}_i + nK)$, with n being the number of sites.

2.2. *Disk-based systems*

The previous algorithm and cost model were based on the assumption that the bipartite graphs fit in main memory. We shall show now that our algorithm can be easily extended to disk-based systems where only a portion of each graph is memory resident. As each bipartite graph is constructed it is dynamically partitioned into subgraphs that are stored on secondary storage, i.e., whenever the buffer allocated to the construction of the graph $BG_{R_i,R_{i+1}}$ becomes full its page(s) is(are) flushed to secondary storage. Thus, each page on secondary storage consists of a subgraph. We observe that these subgraphs are not necessarily disjoint, i.e., they may have crossing edges.

In order to minimize the number of I/O operations that are performed on the bipartite graphs during the backward reduction phase, each graph stored at site S_i is augmented so as to include for every edge (id_{i-1}, id_i) also the page number in secondary storage associated with the tuple containing id_{i-1} at S_{i-1}, to be denoted by $page(id_{i-1})$. Thus, we can view now a bipartite graph BG_{R_{i-1},R_i} as corresponding to a normalized triary relation $BG_{R_{i-1},R_i}(ID_{i-1}, ID_i, PAGE(ID_{i-1}))$. Similarly, the messages sent in the forward and backward phases are augmented correspondingly to include page numbers. Hence, forward messages sent from S_i, the site of BG_{R_{i-1},R_i}, are now of the form $(id_i, a_i, \{page(id_i)\})$, with a_i being the value of the join attribute A_i and $\{page(id_i)\}$ standing for the page number(s) in secondary storage associated with the tuples containing id_i at S_i. Backward messages sent from a site S_i have the format $(id_{i-1}, page(id_{i-1}))$. As we shall see below, using these page numbers we can guarantee that during backward reduction each page of a bipartite graph is read and written back to storage only once.

Since the construction of the bipartite graphs during the forward phase does not guarantee that the subgraphs stored on secondary storage are disjoint, at S_i a particular tuple identifier id_i may appear in a number of distinct pages, connected in each to disjoint (sets of) tuple

Distributed and Parallel Databases 5, 233–269 (1997)

Adaptive Algorithms for Join Processing in Distributed Database Systems

PETER SCHEUERMANN* peters@ece.nwu.edu
Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208

EUGENE INSEOK CHONG† echong@us.oracle.com
New England R&D Center, Oracle Corporation, 1 Oracle Drive, Nashua, NH 03062

Received May 10, 1996; Accepted February 19, 1997

Recommended by: Clement Yu

Abstract. Distributed query processing algorithms usually perform data reduction by using a semijoin program, but the problem with these approaches is that they still require an explicit join of the reduced relations in the final phase. We introduce an efficient algorithm for join processing in distributed database systems that makes use of bipartite graphs in order to reduce data communication costs and local processing costs. The bipartite graphs represent the tuples that can be joined in two relations taking also into account the reduction state of the relations. This algorithm fully reduces the relations at each site. We then present an adaptive algorithm for response time optimization that takes into account the system configuration, i.e., the additional resources available and the data characteristics, in order to select the best strategy for response time minimization. We also report on the results of a set of experiments which show that our algorithms outperform a number of the recently proposed methods for total processing time and response time minimization.

Keywords: distributed query processing, join algorithms, adaptive algorithms, bipartite graphs

1. Introduction

Query processing in distributed database systems often requires the transmission of relations and/or temporary results among different sites via a computer network. Until recently it has been assumed that communication costs were the predominant costs and hence much of the research on query optimization in distributed database systems has concentrated on producing optimal or near-optimal strategies with regard to data transmission costs [1, 3, 5, 6, 8, 12–14].

In order to reduce the communication time, most algorithms for distributed query processing involve three phases [20, 30]: 1) a local processing phase which includes all local operations such as selections and projections, 2) a reduction phase in which an effective sequence of semi-join (and join) operations is further used to reduce the sizes of the relations,

*The work of this author was partially supported by NSF grant IRI-9303583 and NASA-Ames grant NAG2-846.
†The work of this author was performed while he was with Northwestern University.

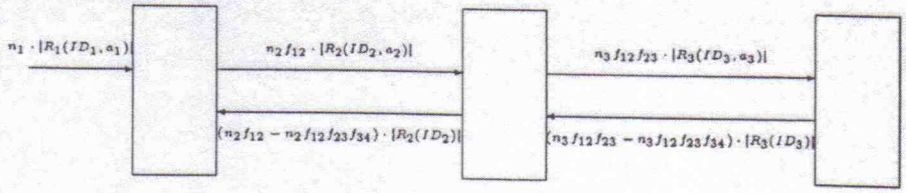

Figure 2. Sizes of transmitted data (PIPE_CHQ).

components. The dominant components of the processing time are the sum $S_{i,f}$, the time to construct the bipartite graphs in the forward reduction phase, and ***GT***, the graph traversal time. Inside the component $S_{i,f}$, the major contribution is the I/O cost for the merge-sort step, while inside ***GT***, the major cost is the I/O time involved in retrieving the target attributes for the tuples in the implicit join. If we denote by $\bar{P}_i$ the average relation size in pages and by K the cardinality of the implicit join, then we obtain that the time complexity of the processing time is bounded by $O(n\bar{P}_i \log_2 \bar{P}_i + nK)$, with n being the number of sites.

2.2. *Disk-based systems*

The previous algorithm and cost model were based on the assumption that the bipartite graphs fit in main memory. We shall show now that our algorithm can be easily extended to disk-based systems where only a portion of each graph is memory resident. As each bipartite graph is constructed it is dynamically partitioned into subgraphs that are stored on secondary storage, i.e., whenever the buffer allocated to the construction of the graph $BG_{R_i,R_{i+1}}$ becomes full its page(s) is(are) flushed to secondary storage. Thus, each page on secondary storage consists of a subgraph. We observe that these subgraphs are not necessarily disjoint, i.e., they may have crossing edges.

In order to minimize the number of I/O operations that are performed on the bipartite graphs during the backward reduction phase, each graph stored at site S_i is augmented so as to include for every edge (id_{i-1}, id_i) also the page number in secondary storage associated with the tuple containing id_{i-1} at S_{i-1}, to be denoted by $page(id_{i-1})$. Thus, we can view now a bipartite graph BG_{R_{i-1},R_i} as corresponding to a normalized triary relation $BG_{R_{i-1},R_i}(ID_{i-1}, ID_i, PAGE(ID_{i-1}))$. Similarly, the messages sent in the forward and backward phases are augmented correspondingly to include page numbers. Hence, forward messages sent from S_i, the site of BG_{R_{i-1},R_i}, are now of the form $(id_i, a_i, \{page(id_i)\})$, with a_i being the value of the join attribute A_i and $\{page(id_i)\}$ standing for the page number(s) in secondary storage associated with the tuples containing id_i at S_i. Backward messages sent from a site S_i have the format $(id_{i-1}, page(id_{i-1}))$. As we shall see below, using these page numbers we can guarantee that during backward reduction each page of a bipartite graph is read and written back to storage only once.

Since the construction of the bipartite graphs during the forward phase does not guarantee that the subgraphs stored on secondary storage are disjoint, at S_i a particular tuple identifier id_i may appear in a number of distinct pages, connected in each to disjoint (sets of) tuple

identifiers from ID_{i-1}. Therefore, during the forward transmission it maybe necessary to transmit, conceptually messages of the form $(id_i, a_i, \{page(id_i)\})$. If a sort-merge is the method of choice for the join to be performed at S_i, then indeed the messages will have the above format. On the other hand, if pipelining is employed, then this information will be transmitted with some slight overhead, i.e., a message of the form $(id_i, a_i, page(id_i))$ needs to be sent for every distinct page at S_i containing tuple identifiers from ID_{i-1} connected to id_i. For example, assume that id_i is connected with two identifiers id_{i-1} and id'_{i-1} and that the corresponding tuples in $BG_{R_{i-1}.R_i}$ are stored on pages p_1 and p_2 respectively. If pipelining is employed, S_i will send the messages (id_i, a_i, p_1) and (id_i, a_i, p_2). Irrespective of the join method used at S_{i+1}, the graph at this site will contain both tuples (id_i, id_{i+1}, p_1) and (id_i, id_{i+1}, p_2) if id_{i+1} is connected to id_i.

The forward reduction phase starts at each site with the construction of the relation $BG_{R_{i-1}.R_i}$ $(ID_{i-1}, ID_i, PAGE(ID_{i-1}))$ and its storage on secondary storage. Next, each page of the bipartite graph is read in order to construct the forward messages, and the page is written back to storage after it has been sorted on attribute ID_i. The sort step is done in order to facilitate the backward reduction and the graph traversal at the query site. We observe here that this sorting step does not incur any additional overhead in terms of I/Os, since the graph had to be stored first on secondary storage in order to obtain the corresponding page numbers necessary for the transmission.

The backward reduction phase at S_i identifies as before all the identifiers id_{i-1} that are not connected to any id_i's and constructs messages of the form $(id_{i-1}, page(id_{i-1}))$. An additional step needs to be performed now, before the actual backward transmission can start, namely all the messages to be sent need to be sorted first by $page(id_{i-1})$. This step is necessary in order to guarantee that at the receiving site, S_{i-1}, each page will be read and written back to storage only once. However, the total amount of memory required at a given site for its outgoing messages is quite small, and this sort can be performed in main memory. In addition, since at the receiving site S_{i-1} each page is sorted on id_{i-1}, a binary search can be performed in order to identify the tuples in the relation $BG_{R_{i-2}.R_{i-1}}$ that need to be eliminated. In our current implementation, these tuples are marked as deleted.

After the backward reduction is completed the size of the bipartite graphs, if compaction were to be executed, could be small enough so that all bipartite graphs fit into main memory at the query site. If this is the case, then Step 4 of the PIPE_CHQ algorithm can be applied the same way, by just ignoring the page numbers. Otherwise, this step is modified and proceeds by interleaving the transmission of bipartite graphs with the construction of temporary relations holding the implicit join tuples. Let us denote by $R'_i \bowtie R'_{i+1} \cdots \bowtie R'_{i+j}$ the implicit join of $R_i, R_{i+1}, \ldots, R_{i+j}$, i.e., the projection of the join on $(ID_i, ID_{i+1}, \ldots, ID_{i+j})$. First, site S_n sends its graph to the query site where the graph is sorted according to $page(id_{n-1})$. Then, we proceed in increasing page number order by joining the tuples in this graph with those in the graph at S_{n-1}. Note that this implicit join can be performed by sending to the query site the pages in the graph of S_{n-1} one at a time and then performing a binary search in the corresponding subgraph of $BG_{R_{n-2}.R_{n-1}}$. After finding the implicit join $R'_{n-1} \bowtie R'_n$ we sort this temporary relation according to $page(id_{n-2})$ and continue in a similar fashion to find the implicit join $R'_{n-2} \bowtie R'_{n-1} \bowtie R'_n$. We repeat this procedure until we obtain $R'_1 \bowtie R'_2 \cdots \bowtie R'_n$.